山东大学
SHANDONG UNIVERSITY

崇新学堂

2024 — 2025 学年第一学期

实验报告

课程名称：Introduction to EECS Lab
实验名称：DL13 - I Walk the Line
学生姓名：胡君安、陈焕斌、黄颢
实验时间：2024年12月19日

DL13 - I Walk the Line

1. Introduction

In this lab, we implement a system for estimating the location of a robot as it moves down a hallway starting from an uncertain location. The key components for our experiment are as follows:

a. Preprocessor which can transform sensor data into inputs for a state estimator

b. State estimator and stochastic state machine with realistic ob-servation and transition models for localization with the 6.01 robots

c. Building a complete sytem demonstrating robot position localization in a one-dimensional world, within the soar simulator

2. Experimental step

Step 1: Understand the implementation of the preprocessor machine

Check Yourself 1

1. What is the internal state of the machine?

   (currentPose, currentSonar)

2. What is the starting state?

   (None, None)

Step 2:

Test the preprocessor on the example from tutor problem Wk.12.2.3 as follows:

a. Run the lineLocalizeSkeleton.py file in Idle.

b. Make an instance of the preprocessor machine, called pp1, using parameters that match the tutor problem: 10 discrete observation values, 10 discrete location values, xMin = 0.0 and xMax = 10.0 (this means that the state width is 1.0 in this example).

c. Do pp1.transduce(preProcessTestData).

d. Make sure the outputs match the ones from the tutor problem.

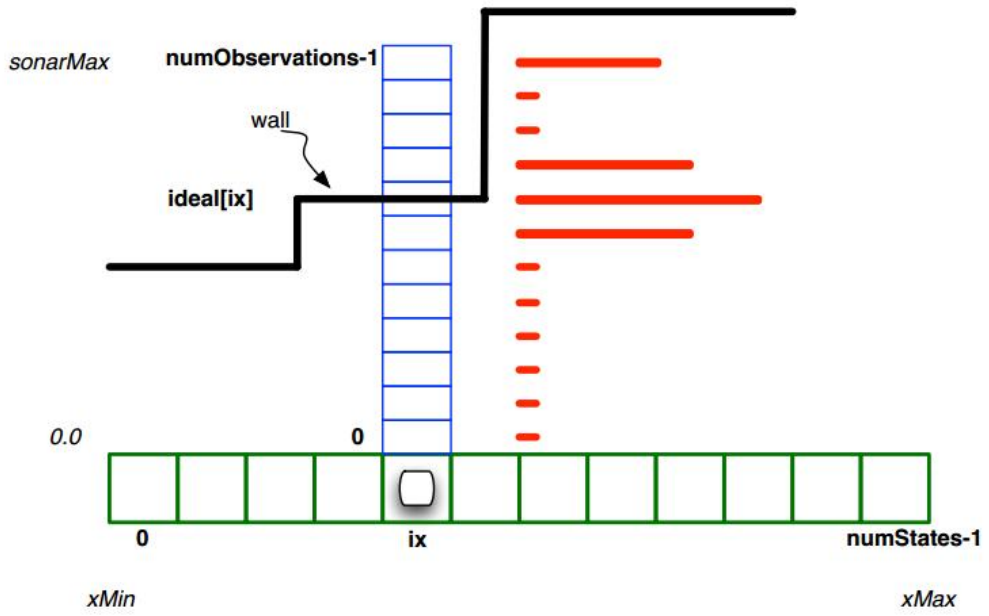
Note that the PreProcess machine prints its output on each step.

```
>>> ppl = PreProcess(10, 1.0)
>>> ppl.transduce (preProcessTestData)
(5, 1)
(1, 5)
[None, (5, 1), (1, 5)]
```

The output

Step 3:

Define startDistribution, which should be uniform over all possible discrete robot locations. Create a uniform distribution over a range of integers with dist.squareDist(lo, hi)



From the picture, we are able to learn that the startDistribution is a squareDist from 0 to numStates-1. As we know, dist.squareDist(lo, hi) returns an average probability distribution from lo to hi-1, so we just have to define lo = 0, hi = numStates.

```
Python
startDistribution = dist.squareDist(0, numStates)
```

Check Yourself 2

Sketch out your plan for the observation model. Be sure you understand the type of the model and the mixture distributions you want to create.

The code is written below.

```
Python
    def observationModel(ix):
        # ix is a discrete location of the robot
        # return a distribution over observations in that
state
        d1 = dist.triangleDist(ideal[ix], 4)
        d2 = dist.DeltaDist(numObservations - 1)
        d3 = dist.squareDist(0, numObservations)
        return dist.MixtureDist(dist.MixtureDist(d1, d2,
0.9), d3, 0.95)
```

Step 4:

Implement the observation model and test it to be sure it's reasonable. It doesn't need tomatch the histogram in the figure exactly.

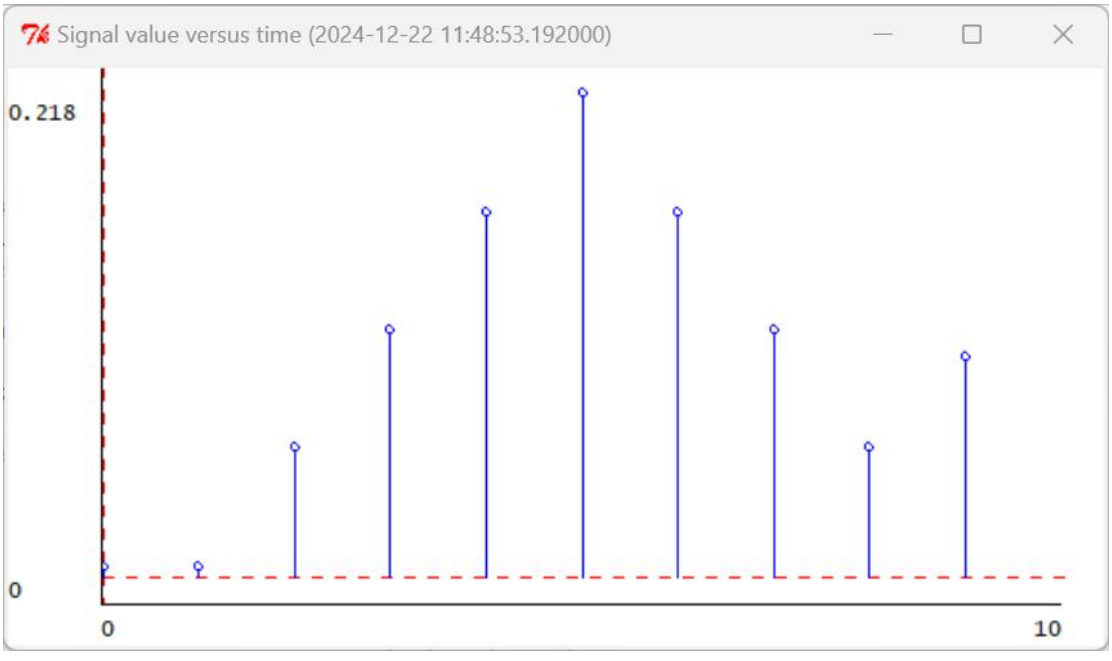
For debugging, we create

```
Plain Text
model = makeRobotNavModel(testIdealReadings, 0.0, 10.0, 10,
10)
```

Then we get the observation conditional probability distribution through:

```
Plain Text
model.observationDistribution
```

We debug the distributions by plotting them.



Q

What does the 7 stand for here?

When calling observationModel (7), 7 represents a discrete position index of the robot.

Step 5:

Now, make a model for the case with 100 observation bins, instead of 10.

```
Plain Text
model100 = makeRobotNavModel(testIdealReadings100, 0.0,
10.0, 10, 100)
```

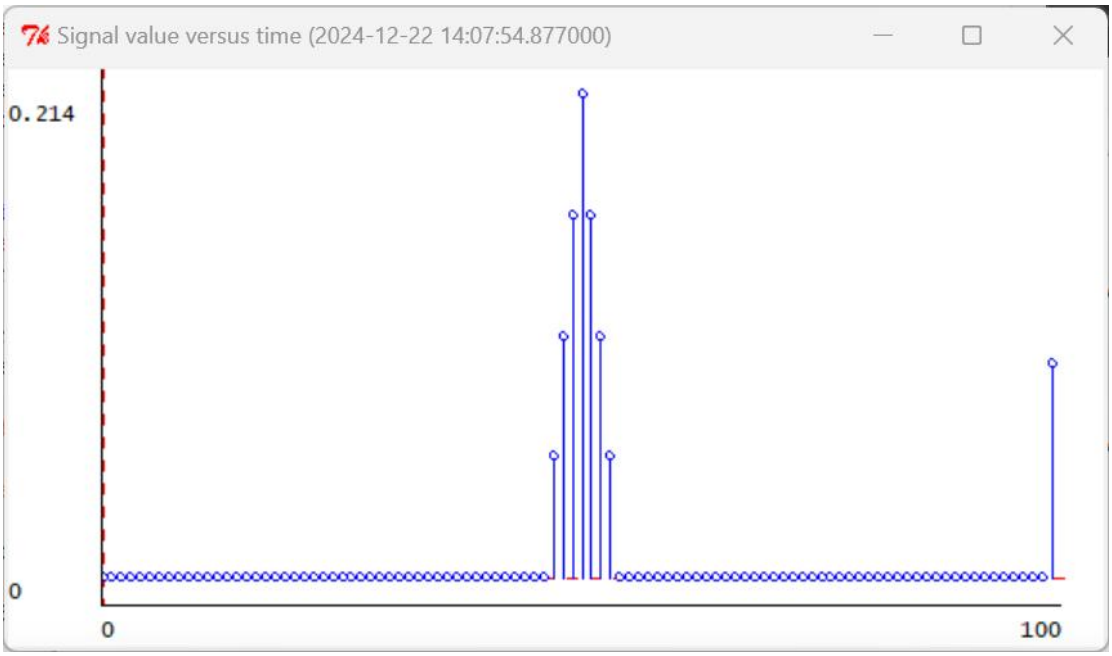
Then plot the observation distribution for robot location 7 in model and model100.

Checkoff 1

Wk.13.2.1: Show your observation distribution plots to a staff member and explain what they mean.

The picture of our observation distribution plots is laid below.

Compared to Model 1, their trends are completely identical -- increasing first before decreasing. Then we gain the maximum distribution in a certain interval.



Check Yourself 3

Sketch out your plan for the transition model. Be sure you understand the type of the models and the distributions you want to create.

The code is written below.

The transition model is built by defining transitionModel(a) which returns transitionGivenState(s). In it, a mixture of triangle and uniform distributions is created. Then it's used for the stochastic SM.

```
Python
    def transitionModel(a):
        # a is a discrete action
        # returns a conditional probability distribution on
the next state
        # given the previous state
        def transitionGivenState(s):
            transUniform =
dist.UniformDist(range(numStates))
            return
dist.MixtureDist(dist.triangleDist(util.clip(s+a, 0,
numStates-1), 3, 0, numStates-1),
                             transUniform, 0.9)
        return transitionGivenState
```

Step 6:

Implement the transition model and test it to be sure it's reasonable. Create a

ssm.StochasticSM, and then get the transition model (which is a procedure that returns a conditional probability distribution) like this:

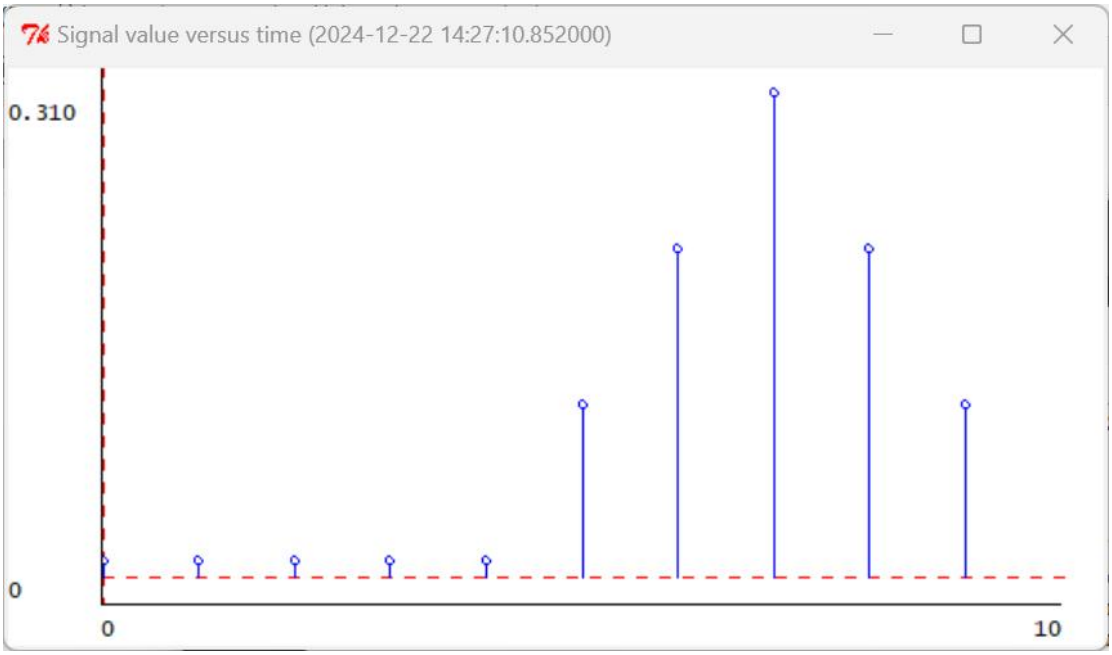
```
Plain Text
model=makeRobotNavModel(testIdealReadings, 0.0, 10.0, 10,
10)
model.transitionDistribution
```

Q

If transitionModel is your transition model, write down transitionModel(2)(5) (this is an instance of DDist). What do the 2 and 5 stand for here?

The transitionModel (2) (5) is used to calculate the probability distribution of the

resulting state that the robot may reach after performing a specific action (2) in a specific initial state (5).



Step 7:

Now we'll put the two modules we just made together and be sure they work correctly. Use sm.Cascade to combine an instance of your PreProcess class and an instance of the seGraphics.StateEstimator class, which is given your ssm.StochasticSM model, using 10 discrete observation values, 10 discrete location values, xMin = 0.0, and xMax = 10.0. Call this machine "ppEst".

```
Python
ppEst = sm.Cascade(PreProcess(10, 1),
seGraphics.StateEstimator(makeRobotNavModel(testIdealReading
s, 0.0, 10.0, 10, 10)))
ppEst.transduce(preProcessTestData)


# >>>
# (5, 1)
# (1, 5)
```

Check Yourself 4

Do ppEst.transduce(preProcessTestData). Compare the result to the belief states in Wk.12.2.3. Remember that you are now assuming noisy observations and noisy actions. Are your results consistent with the ones you found in the tutor?

Noisy observations and noisy actions are tantamount to altering the value of val during state transitions, which represents the distance that sonar inputs to the state machine. This leads to different judgments of the trolley. In other words, the sonar output is inaccurate, meaning the judged distance by sonar is incorrect. Therefore, noise can readily cause the system to go out of control.

Step 8:

Now, we'll put all the machines together to make a behavior that can control the robot.

In your lineLocalizeSkeleton.py file, implement the procedure makeLineLocalizer with the arguments shown above; it should construct a complete robot behavior, as outlined in the architecture diagram, whose inputs are io.SensorInput instances and whose outputs are io.Action instances.

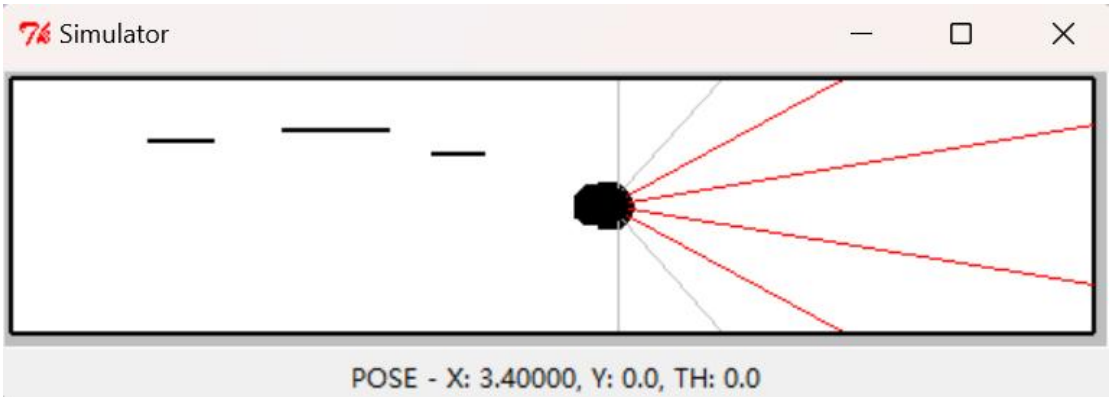
```
Python
def makeLineLocalizer(numObservations, numStates, ideal, xMin,
xMax, robotY):
    width = (xMax - xMin) / float(numStates)
    preprocessor = PreProcess(numObservations, width)
    estimator =
seGraphics.StateEstimator(makeRobotNavModel(ideal, xMin, xMax,
numStates, numObservations))
    driver = move.MoveToFixedPose(util.Pose(xMax, robotY, 0.0),
maxVel = 0.5)

    return sm.Cascade(sm.Parallel(sm.Cascade(preprocessor,
estimator),driver),
                      sm.Select(1))
```

Step 9:

Start soar and run behavior using lineLocalizeBrain.py in the world worlds/oneDdiff.py.

Use the step button in soar to move the robot two steps:



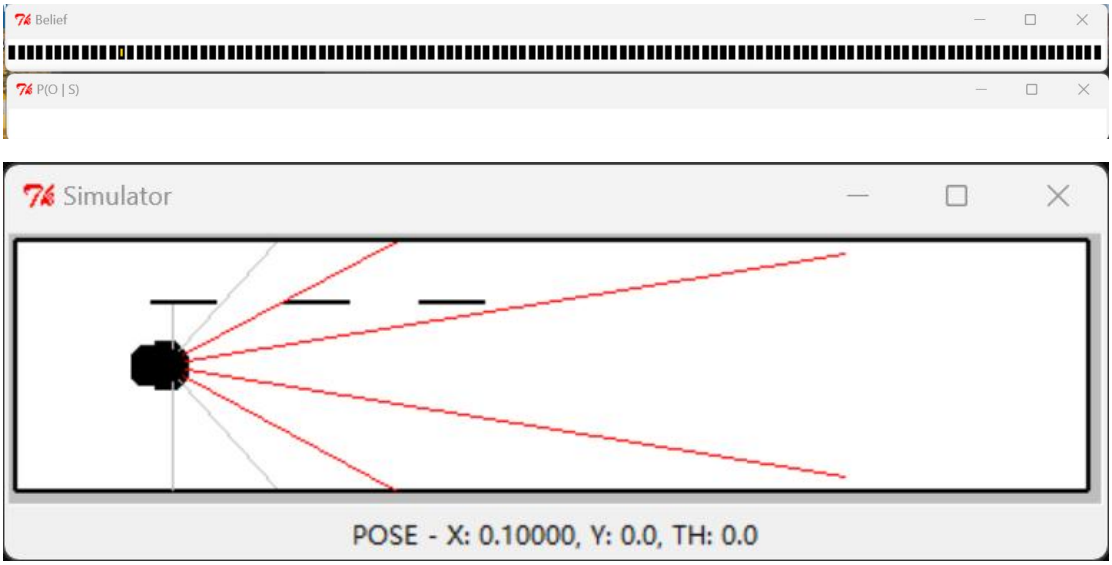

```
Ordinary output will appear in this window.
***Loading Brain***
'C:/Users/dnars/Desktop/designLab13-code/lineLocalizeBrain.py'
setting labPath to C:\Users\dnars\Desktop\designLab13-code
(5, 1)
(1, 5)
(5, 1)
(1, 5)
World file is oneDdiff.py
Successfully loaded brain file
'C:/Users/dnars/Desktop/designLab13-code/lineLocalizeBrain.py'
(7, 2)
(7, 2)
```

```
(7, 2)
(7, 2)
(7, 2)
(7, 2)
(7, 2)
(17, 2)
(17, 2)
(17, 2)
(17, 2)
(17, 2)
(9, 2)
(9, 2)
(9, 2)
(9, 2)
(9, 2)
(9, 2)
(9, 2)
```

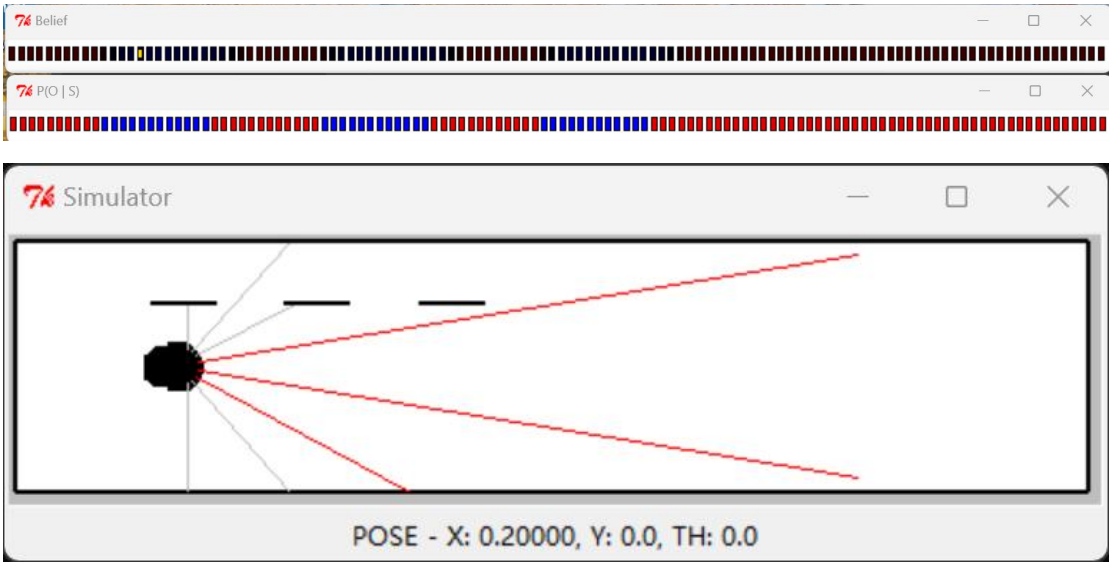
```
(9, 2)
(17, 2)
(17, 2)
(17, 2)
(5, 2)
(5, 2)
(5, 2)
(5, 2)
(17, 2)
(17, 2)
(17, 2)
(17, 2)
(17, 2)
(17, 2)
(17, 2)
(17, 2)
```

Step 10:

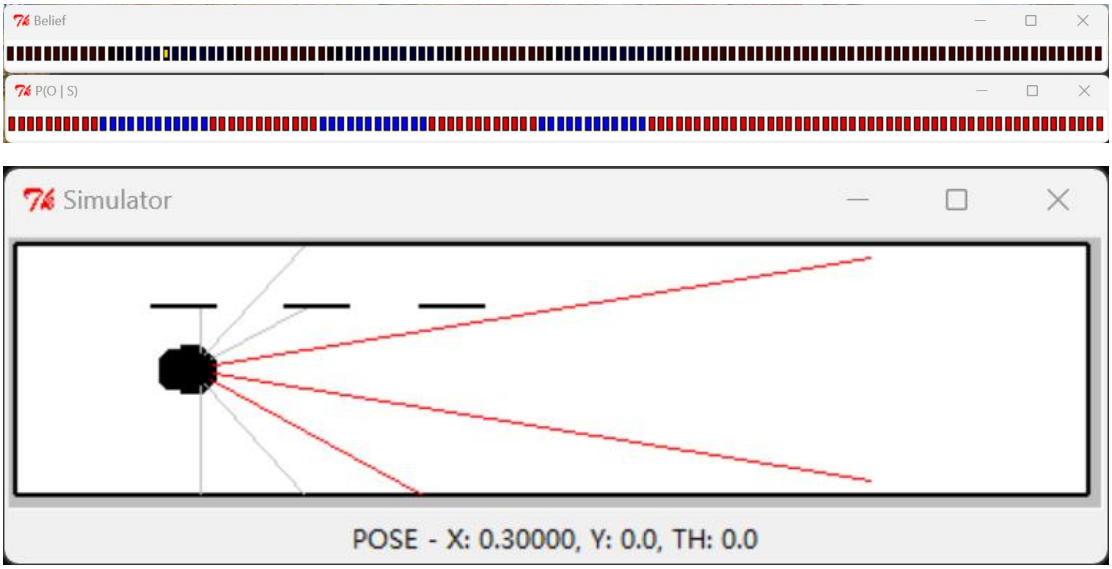
After 1 step:



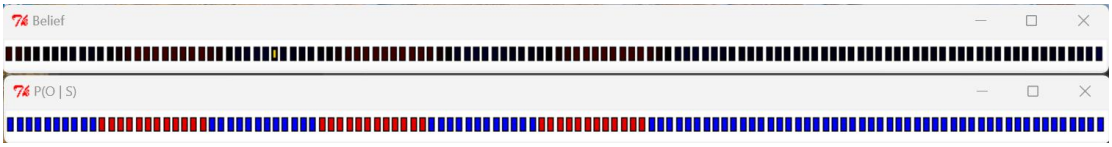
2steps:

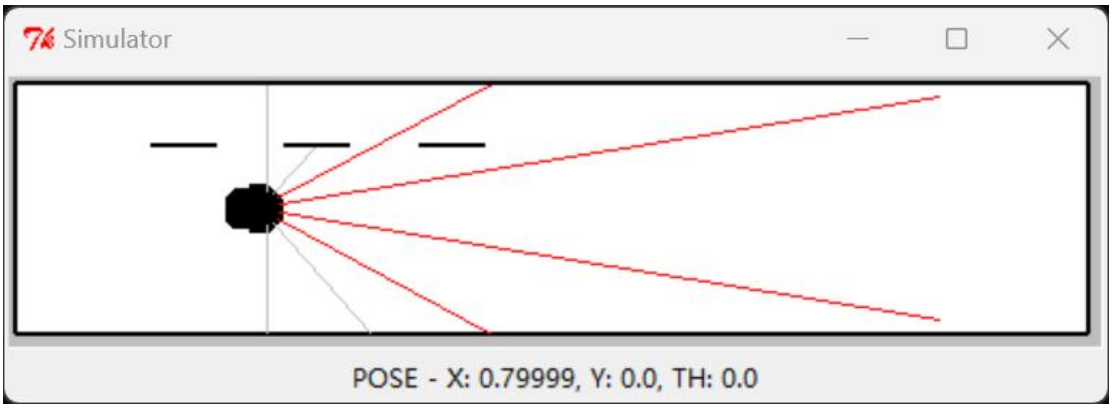


3steps:



8steps:






```
Ordinary output will appear in this window.
***Loading Brain***
'C:/Users/dnars/Desktop/designLab13-code/lineLocalizeBrain.py'
setting labPath to C:\Users\dnars\Desktop\designLab13-code
(5, 1)
(1, 5)
(5, 1)
(1, 5)
World file is oneDreal.py
Successfully loaded brain file
'C:/Users/dnars/Desktop/designLab13-code/lineLocalizeBrain.py'
(7, 2)
(7, 2)
(7, 2)
(7, 2)
(7, 2)
(17, 2)
(17, 2)
```

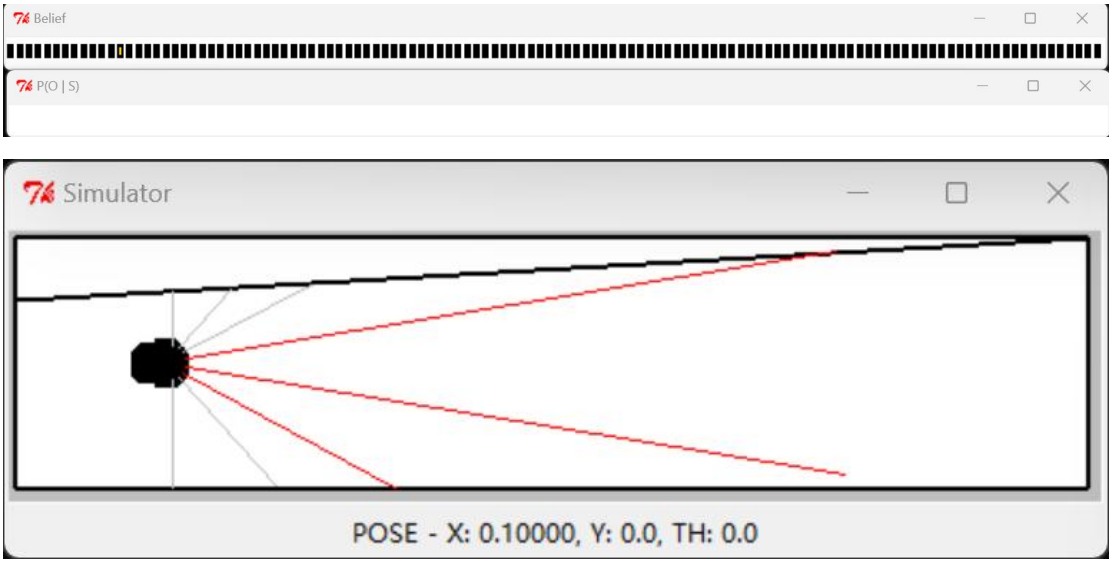
Q

Now run your behavior in the world oneDreal.py (you will need to edit the line in lineLocalizeBrain.py that selects the world file, as well as select a new simulated world in soar). What is the essential difference between this world and oneDdiff.py?

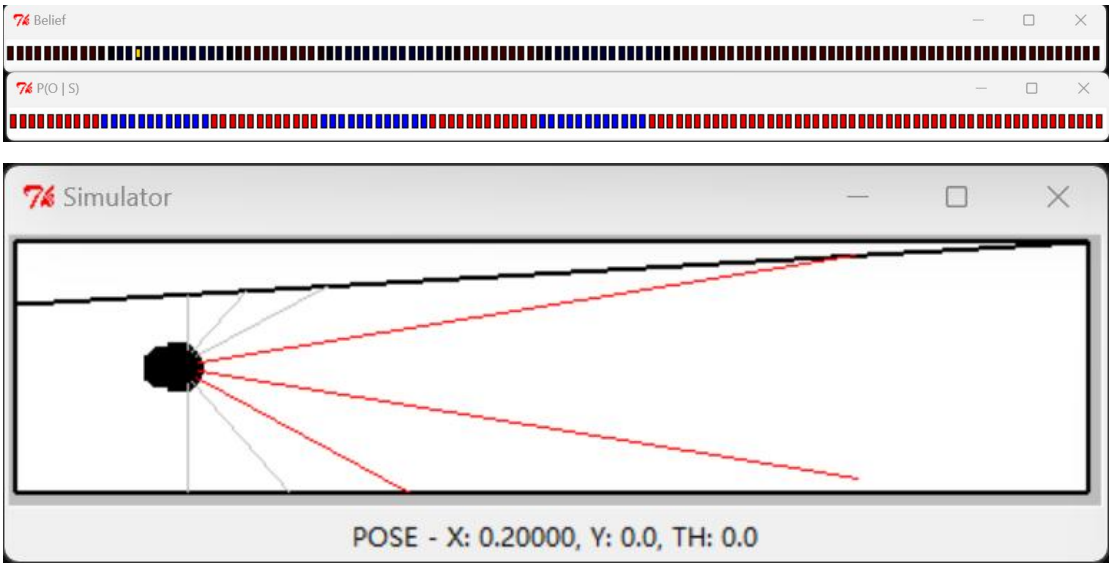
The fundamental difference between the two maps is that the distance between the three walls of the latter and the boundary of the map is the same, which means the positioning function of robots inferior to that in the first one--oneDdiff world.

Step 11:

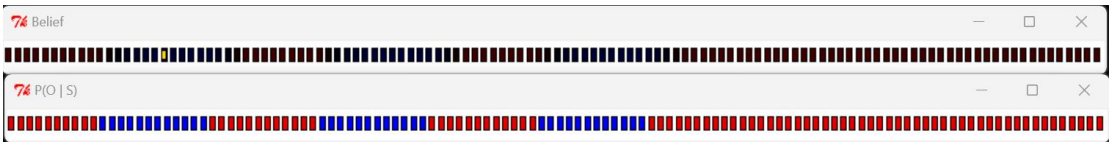
1step:

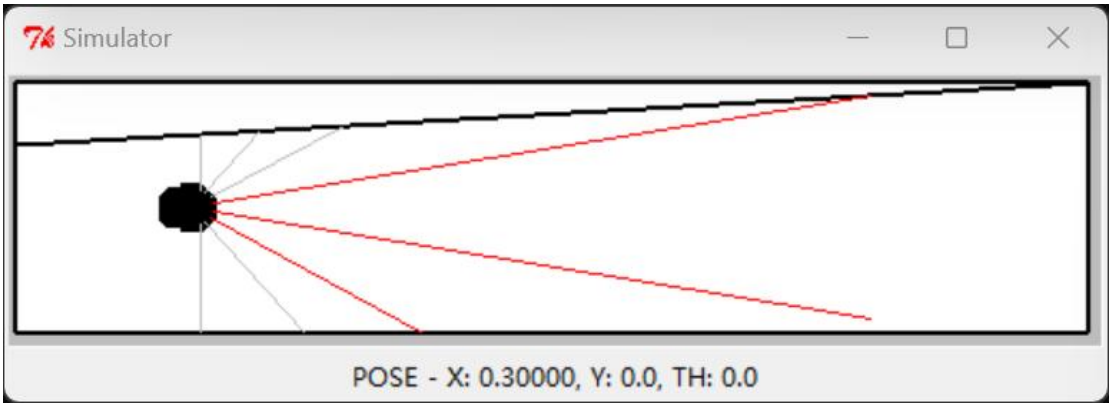

Belief
P(O | S)
Simulator
POSE - X: 0.10000, Y: 0.0, TH: 0.0

2:

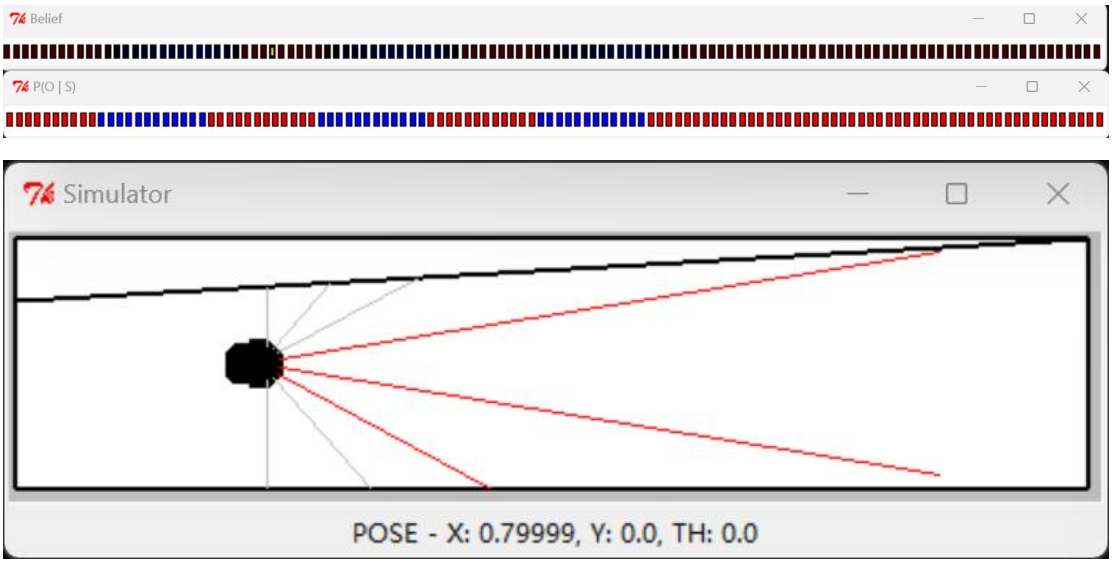

Belief
P(O | S)
Simulator
POSE - X: 0.20000, Y: 0.0, TH: 0.0

3:


Belief
P(O | S)



8:




```
Ordinary output will appear in this window.
***Loading Brain***
'C:/Users/dnars/Desktop/designLab13-code/lineLocalizeBrain.py'
setting labPath to C:\Users\dnars\Desktop\designLab13-code
(5, 1)
(1, 5)
(5, 1)
(1, 5)
World file is oneDreal.py
Successfully loaded brain file
'C:/Users/dnars/Desktop/designLab13-code/lineLocalizeBrain.py'
(8, 2)
(8, 2)
(8, 2)
(9, 2)
(9, 2)
(9, 2)
(9, 2)
```

Q

Now run your behavior in the world oneDslope.py, without changing the world file

selected in the brain. This will mean that the robot thinks it is in the world oneDreal.py, and has observation models that are appropriate for that world, but it is, instead, in an entirely different world. What happens when you run it? What do the displays mean?

When the distance between the robot and the wall is the distance in the oneDream world, the robot will locate its position based on the conditions of oneDream. If not, it will fail to locate itself.

Checkoff 3

Wk.13.2.3: Demonstrate your running localization system to a staff member. Explain the meanings of the colors in the display windows and argue that what your system is doing is reasonable. Explain why the behavior differs between oneDreal and oneDdiff. Explain what happens when there is a mismatch between the world and the model.

The reason was explained in the previous steps

3. Summary

- In this experiment, we delved into the diverse motion states of robots within different worlds. Specifically, a system was implemented to estimate the position of a robot as it travels along a corridor, initiating from an uncertain position.
- The system incorporated a transition model, such as the one defined by the `transitionModel` function. This function, which takes a discrete action as input and returns a conditional probability distribution on the next state given the previous state, was crucial in simulating the robot's state transitions. Additionally, a stochastic state machine (`ssm.StochasticSM`) was constructed using the start distribution, the transition model, and the observation model.
- The observation model might account for factors like noisy observations, which could be analogous to the sonar input with potential inaccuracies. Through this comprehensive experimental setup and the utilization of these key components, we have successfully laid a solid foundation for the final, more advanced and complex experiment that might involve enhanced state estimation, more refined action planning, and improved robustness against uncertainties and errors.